Algorithms for Software Development Version Control and Change Detection

R. C. Tausworthe
DSN Data Systems Section

This article describes simple computer algorithms for processing source program and text data files in order to extract change detection, version control, version history, and current status information easily. These algorithms presuppose that it is possible to attach to each record of the source files a 6-character code, placed within delimiters that will cause the compiler, or other using program, to ignore this code field. The code contains a 2-character code for a character-by-character position-sensitive checksum of the record, another for the record number in the file, and a third for the date on which the encoding took place. Once the source file has been thus encoded, it is possible to detect the following transactions on the file since the most recent version coding: (a) addition of new records (having no version code); (b) modification of existing records; (c) deletion of a number of records; (d) movement and/or duplication of existing records; and (e) modification and duplication of records. In addition, it is possible to extract a version history of the number of records created or modified by date. A special file listing program is described which prints the file records without showing the version codes, but places a "change bar" at the right margin whenever a change is detected. The program also provides a list of changed pages and a version history.

I. Introduction

During the development and maintenance phases of software tasks, it is extremely convenient to have some way of distinguishing in what way a particular version of the product differs from its ancestor versions. In this way, those persons making and monitoring the changes in the product can more readily focus on the differences made since a particular proven benchmark. When one is given a new release of a large document or program for review, for example, one must be able to locate what is different about *this* version, as opposed to the last one, in order to avoid the needless expense of rereading the entire document.

Managers concerned with implementation status may be interested in knowing the rate at which the product is being developed, as well as other statistics of the process, such as the amounts of new code, modified code, and reused, unmodified code contributing to a particular effort.

It is often important to control the issuance of new versions, so that only a known, prescribed, and contracted capability is released to customers and utilized by them.

Such change identification, version history, status, and version control processes are virtually infeasible for large computer programming projects without computerized aid. Some techniques that have been employed in the past include compile-time dating of the listing and source file, usage of file comparison programs, and special annotations of the source made by a text editor while producing the program text. All of these techniques are useful, but have their limitations, as does the technique described in this article. Compile-time dating, for instance, suffers by only telling when the program was last compiled, not what is different about the program. File comparison programs are generally slow and can only identify relatively small changes between the two versions of the source. Annotations placed in the source by a text editor, usually limited to sequence numbers of the records, or perhaps a version number or date of last update of the entire file, do not provide for identification of modifications made on individual records, nor do they distinguish which lines have appeared in which versions of the software.

This article describes a combination of techniques yielding simple computer algorithms for change identification and version history that are efficient in file storage requirements, execution speed, programmer overhead, and operations. These algorithms can be applied whenever it is possible to attach to each record of the source files a 6-character code, placed within delimiters that will cause the compiler, or other using program, to ignore the code field. The code contains a 2-character code for a character-by-character position-sensitive checksum of the record, another for the record number in the file, and a third for the date on which the encoding took place. Once the source file has been thus encoded, it is possible to detect the following transactions on the file since the most recent version coding: (a) addition of new records (having no version code); (b) modification of existing records; (c) deletion of a number of records; (d) movement and/or duplication of existing records; and (e) modification and duplication of records. In addition, it is possible to extract a version history of the number of records created or modified by date. A special file listing program is described which prints the file records without showing the version codes, but places a "change bar" at the right margin wherever a change is detected. The program also provides a list of changed pages and version history.

The scenario of operations is as follows (see Fig. 1): The principal activities associated with software development and sustaining involve the programmer interacting with a text editor building and modifying program and documentation source files. At regular intervals, these source files are submitted to a version encoding program that checks each line for correct checksum and then resequences the file. When the checksum is correct (i.e., matches the checksum in the code supplied in an earlier run), the date is not changed. If the checksum is in error, the current date is inserted in the date code. Thus, after each run of the encoding program, the source file has a correct checksum code, correct sequence number code, and applicable date for each line in the file.

Subsequent editing of the file may create new lines, delete old ones, modify characters, insert characters, move lines, or copy (reuse) existing lines. The combination of the three elements of the line code permits all of these transactions to be detected, including identification of changes made since a given status date.

Each time the source file is checksummed, information as to which lines have been added, which modified, how many were deleted, etc., is lost. However, version history information is retained, as it is possible to tell which lines were redated at each checksumming.

II. Source File Structure

The source file structure after the version coding operation is described in the following schematic logic structure:

```
Source file
  *Physical line
      *Data segment
       Version code
         Left delimiter
         Checksum code
           2 characters
         Sequence number code
           2 characters
         Date code
           2 characters
         o Right delimiter
```

The annotations (* and o) in this logic refer to iteration and alternative structures, respectively. Indentation corresponds to refinement of the item definition. This structure thus defines the source file as an iteration of physical lines, each made up of a number (perhaps zero) of data segments, followed by a version code. The version code, in turn, consists of a left delimiter, checksum, sequence number, date, and perhaps a right delimiter (depending on the compiler or program using the file).

The delimiters will generally need to be file-dependent. For program source files, these delimiters will be chosen to make the code appear as a comment to the compiler, perhaps with a character or two thrown in for easy recognition that this comment is the version code, not an ordinary comment.

Other files may require special handling procedures, such as special input subroutines to fetch input records and strip out the version codes, or a preprocessing pass of the source file to create a dummy file without the codes for access by the program.

III. Generating the Two-Character Codes

The two characters used in the three code elements could be any printable members of the ASCII set, i.e., ranging from space (character 32) through the tilde (character 126). Control characters (0 through 31) are apt to cause sensitivities within the using program(s), especially compilers, and are thus omitted here. Also, characters following "z" in the ASCII set may be set aside for use only in the delimiters, leaving 91 characters for the code elements.

Care must be taken in generating the version code not to generate character sequences sensitive to the using program. For example, if a Pascal program were being version encoded using comment delimiters "(*"and"*)" around the code, then care would have to be taken to ensure that the characters "*)" do not appear within the version code. Pascal has alternate comment delimiters "{"and"}", ASCII characters 123 and 125, which may be used to obviate this problem. However, PL/I programs will have problems with "/*"and""*/". If the algorithms herein are to be applied to Pascal programs not permitting the alternate delimiters or to PL/I programs, then the version code should omit the offending combinations. For the remainder of this article, we shall assume that the full 91-character span can be used.

The combination of two such characters is sufficient to span a range of 91 X 91 = 8281 alternatives. The checksum of characters on the line, discussed later, will lie randomly within the interval (0, 8281). The probability that a randomly modified line will have the same checksum as was recorded before the change will be 1/8281 = 0.00012; this low figure is probably sufficient for most change identification purposes. The particular checksumming algorithm presented is one which makes almost all of the usual types of modifications to a line detectable within the probability stated above.

The limit of the sequence number to 8281 limits the file size to as many records; this probably does not pose much of a problem, as programs this large are usually segmented for compilation purposes, and documents this large are similarly segmented for ease in processing.

The correspondence between a number N and its representation by the two code characters C1 and C2 is defined by

$$C1 = \mathrm{chr}(\mathrm{ORD}('\ \ ') + N/91)$$

$$C2 = \mathrm{chr}(\mathrm{ORD}('\ \ ') + (N \text{ modulo } 91))$$

where chr(n) is the ASCII character having collating sequence number n and ORD(C) is the collating sequence number for the character C. The ASCII collating sequence number for space is 32.

Similarly, given a code with the two characters C1 and C2, the corresponding number N is

$$N = 91 * [\mathrm{ORD}(C1) - \mathrm{ORD}('\ \ ')] + \mathrm{ORD}(C2) - \mathrm{ORD}('\ \ ')$$

$$= 91 * \mathrm{ORD}(C1) + \mathrm{ORD}(C2) - 92 * \mathrm{ORD}('\ \ ')$$

A. The Date Code

The conversion from a date having day number d in month m of year 1980+y into the date code number D is defined by

$$D = (d-1) + 31 * (m-1) + 372 * y$$

This date number can be encoded into two characters as described above.

When the two-character date code has been retrieved from a line and converted to a date number D, the day of the month will be computed as 1+(D modulo 31), the month as 1+(D/31 modulo 12), and the year as 1980 + (D/372). In these computations, and indeed, throughout the remainder of this article, integer division is assumed (remainder discarded) whenever "/" appears in an expression. Although this date encoding appears to use up 372 days per year (i.e., 6 or 7 unused numbers each year), it is still sufficient to carry the version code through 8281/372=22.26 years, or until March 5, 2002.

B. The Checksum Code

The line characters to be checksummed, call them $C_1, \ldots, C_N$, are combined by the following algorithm:

Each character C_i is combined with the line checksum L accumulated so far to obtain a new value by bit-wise "exclusive-or" (xor) and "and" operations. First set $L = 0$; then for each of the C_i,

$$\text{set } L = L \text{ xor } 2*(L \text{ and } 16383),$$

and then

$$\text{set } L = L \text{ xor } (L/16384) \text{ xor } \mathrm{ORD}(C_i))$$

Finally,

$$\text{set } L = (L \text{ modulo } 8281)$$

This computation simulates the driving of a maximum-period linear feedback 15-element finite-state machine with the character values from the input line. Values of the checksum lie in the range 0-8281. The iterations above, when given only a single non-null character and nulls thereafter, generate a pseudo-random sequence which repeats only after $2^{15}-1$ = 37,767 such characters have been input. The combination when characters are non-null is a convolution of the input sequence with the linear pseudo-random response of the finite-state machine. This method tends to randomize the checksum values in an extremely character-sensitive way, so that the chances of any modification to an input line falling in the same equivalence class of input lines having the same checksum is 1/8281 = 0.00012.

The recursion relation above corresponds to the maximum-period linear shift-register device having the primitive characteristic polynomial over GF(2)

$$f(x) = x^{15} + x + 1$$

The algorithm for utilization of this polynomial to generate pseudo-random numbers is given in [Ref. 1]. The operations are to shift the checksum left one bit, filling a 0 on the right, and xor with the unshifted value. Then xor this intermediate result with a right shift of itself by (15-1)=14, filling 0's in on the left. For further information regarding the properties of shift-register sequences, see Ref. 2.

C. Sequence Number Code

The sequence number code is a straight translation of the record sequence number, starting at zero, into the two-character code, and vice-versa.

IV. The Checksum Program

This section describes the design of a prototype program written to produce the updated version-encoded source file. For convenience, it will be referred to herein as "CHECKSUM".

Some basic assumptions about the files processed by the program and the processing done by the program are:

- Three types of version codes will be accommodated:
  - files requiring only a left version code delimiter
  - files requiring both left and right delimiter
  - files in which version codes are restricted to certain columns on the input line
- It will be possible to detect which of these version code types is to be used from the file name, say by the file name extension; alternately the program could accept this information from the user of the program.
- Source (program) files may have embedded comment lines containing design information expressed in a Program Design Language (PDL). Such lines are distinguished by the presence of an "extraction code" at the beginning of the line.
- PDL lines and non-PDL lines are to have separate, independent sequence numbering so that if the PDL is extracted, the numbering is still correct.
- PDL and non-PDL lines may have different delimiters, if accommodated by the using program.
- Placement of the version code on an output line may be designated to appear in a certain column on the line if desired.

The main procedure of the program is described in the program design language of Fig. 2; the structure of the version code parameter lookup table is presented in Fig. 3; the extraction algorithm for retrieving version code information from the input line is given in Fig. 4; and the method for checksumming and reformatting of the output line is shown in Figs. 5 and 6.

V. The Version Code Removal Program

The CHECKSUM program must detect whether an input line has a version code on it, and whether it is a PDL variety or not, so that the code can be removed and another reattached in its place. However, in some applications, it may be necessary to strip out the version codes from an entire file before it can be properly processed by its using program, or when a file is to be completely reencoded.

The algorithm given in Fig. 4 has been used in such a program, called STRIP, which removes the version codes from an entire file. The algorithm has been only slightly modified, shortened because the BEGINNING, SEQUENCE_NUMBER, and SEQUENCE_CODE variables are not needed.

VI. The File Listing Program

The file listing program, called PRINT, is similar to the CHECKSUM program in overall structure, except that a "change bar" is reattached to the output line, rather than the version code. Also, the PRINT program keeps track of what sequence numbers have appeared in the file, and therefore knows when deletions and duplications have occurred, as well as the number of lines which have undergone such transactions with the text editor since the most recent processing by CHECKSUM.

Besides the name of the file to be processed, a status date is given to the PRINT program; all lines of the input file having a date code after this status date will be identified on the listing. The change bar applied to the output is chosen to distinguish what kind of change has been detected:

	Change bar symbol	Detected change		
		Checksum	Sequence number	Date
No change	(None)	OK	OK	= < Status date
Change since status date	[ ]	OK	OK	> Status date
Added line	[A]	None	None	None
Sequence number went backwards	[B]	–	< Previous	–
Deleted line(s)	[D]	–	> Previous + 1	–
Modified line	[M]	Bad	OK	–
Reused line	[R]	OK	Duplicate	–
Reused and Modified	[RM]	Bad	Duplicate	–

Whenever [B] or [D] situations are detected, a blank line is printed with these change bars; then the input line checksum and date codes are checked as in the table above for possible application of either (none), [ ], or [M]. The PRINT program records the page number each time a change is detected. A list of changed pages is then output to assist readers of the documentation. Separate counts of the numbers of input lines falling into each of these categories (except [B] and [D]) are maintained as the input file is processed. The total number of deletions is determined after the entire file has been processed.

In addition, each time an input line is processed, the version history is updated, as follows: The date on the line (or current date if the line has none) is found (or inserted, if not found) in the VERSION_DATE field of the VERSION_HISTORY table, and the corresponding NUMBER_OF_TIMES entry in the table is incremented. After the entire file has been processed, the version history of dates vs. number of lines is printed.

The algorithm for detection of deletions and duplications of records makes use of a list of pairs of numbers corresponding to intervals of sequence numbers *not* yet found up to the current input line. Initially, the PRINT program starts with the list containing only the pair (0, INFINITY). At any time the list will contain a number of such pairs signifiying ranges of sequence numbers not yet seen. For example, if at one point the list contained the pairs (5, 8), (10, 10) and (12, INFINITY), then only the sequence numbers 0, 1, 2, 3, 4, 9, and 11 will have been seen so far.

Whenever this list is checked against the current input line sequence code (converted to a number), the sequence number may be found not to appear anywhere within the intervals of any of the pairs in the list; in this case, the sequence number is a duplication, it has already been seen. However, in the usual case, the sequence number has not been seen yet, having been determined to lie within the interval specified by a particular pair in the list. Then one of three things takes place:

(1) If the sequence number is the lower limit of the pair, this lower limit is incremented by 1

(2) If the sequence number is the upper limit of the pair, this upper limit is decremented by 1

(if in either of these two cases the lower limit afterwards exceeds the upper limit, the pair is deleted from the list)

(3) If the sequence number is between the limits (low, high) of the pair, the pair is split into two pairs:

(low, seq_no−1), (seq_no+1, high)

When all the records of the input file have been processed, the number of deletions may be computed as follows: the last list pair (last+1, INFINITY) is discarded; the remaining pairs identify the gaps in sequence numbers of the input file. These intervals may be printed out, if desired (PRINT does not, however). The total number of deletions is the (high-low)+1 value of each list pair summed over all remaining list pairs.

Since the input file may be a source program containing embedded PDL statements sequence numbered separately, it is necessary for the PRINT program to maintain separate SEQUENCE_NUMBER counters and occurrence lists for the two types of records.

VII. Application Problems

Several concerns may occur to those considering the application of version codes to their source files. These concerns include

(1) Expansion of file size due to version code overhead.

(2) Sensitivity of the using program(s) to the version code.

(3) Effort demands and discipline required of user to update the version codes periodically.

(4) Effort and nuisance of having to remove the codes before submitting the file to a sensitive using program.

(5) Nuisance in seeing meaningless version codes on compiler output listings, even when compiler is insensitive to the codes.

(6) Impracticality of applying the version codes to some kinds of packed text files, such as commonly created by some word processors, that are free-form and not line-oriented.

The applicability of the technique described in this article is therefore not universal, as it stands. Some of the difficulties may be slaked by the advantages of a version control and change identification capability, such as a slight file size overhead, the discipline of update and archival, etc. Others, such as the sensitivity of the using program(s) to the appearance of the version code, present real problems.

The technique, as it stands, *is* very useful in the following cases:

(1) For input text files of any kind whenever the using program(s) may be adapted, or written from the outset, to ignore the version codes. Programmers can be supplied with standard library subroutines to handle input of fundamental data types.

(2) For program source code when the programming language permits a comment to end the line, signalled by a comment left delimiter only.

(3) For program source code when the programming language permits a comment to end the line, using both left and right comment delimiters. This usually requires that the programmer not put comments in the program that extend across physical line boundaries.

The algorithms herein may be modified to accommodate the last of these cases when comments do extend over physical line boundaries. Such modification would demand only that the detection and reattachment procedures for the version code be extended to sense whenever the version code was being inserted within a comment or not, and to choose alternate delimiters as appropriate.

Should the method described above prove altogether infeasible because of the limitations listed, it may prove worthwhile to investigate the reasonability of modifying the operating system to reserve positions within each file record for version code usage, with special access functions to retrieve and deposit these codes. Existing programs using a file would then be unaware of the version code altogether, as it would never be delivered by the operating system when file data is requested. There are some obvious problems that one immediately encounters in such a proposed solution, such as how the operating system accommodates an existing, unmodified text editor in maintaining the correspondence of input file codes with rewritten output file codes. Such an investigation seems premature at this point, however.

VIII. Prototype Demonstration Results

To demonstrate the methods of this article, the program text of Fig. 2 was chosen for example. It was checksummed, modified, and rechecksummed repeatedly to simulate the growth and correction of a software design. Then it was modified without rechecksumming, so that additions, deletions, modifications, and duplications would be made more evident. Then it was printed out using PRINT, so that changes and version history would be shown.

Figure 2 shows the original text; Fig. 7 shows the encoded text, using left and right delimiters "{" "and"} ", respectively; and Figs. 8 and 9 show the output of the PRINT program, which illustrates the appearance of change bars, the change statistics, and the version history printout. The list of changed pages is not shown, since the example is but one page long.

IX. Conclusion

The technique given here is a useful and feasible method for providing source file change detection and version control measures for many types of program, documentation, and configuration data files. As the application of such techniques becomes more widespread, one will find that the sensitivities to version encoding will certainly decrease. Operating systems will accommodate the codes automatically; compilers and other system software will expect such codes to be present; programmers will come to expect the system to provide version statistics services without incursion on their productivity; and managers and maintenance personnel will be able to perform their functions more effectively because of the better visibility into exactly what the extent and content of changes to a software package have been in any given release.

References

1. Tausworthe, R. C., "Random Numbers Generated by Linear Recurrence Modulo 2," *Math. Comp.*, Vol. XIX, No. 90, pp. 201-208, 1965.

2. Golomb, S. W., et al., *Shift-Register Sequences*, Holden-Day, Inc., San Francisco, Calif., 1967.

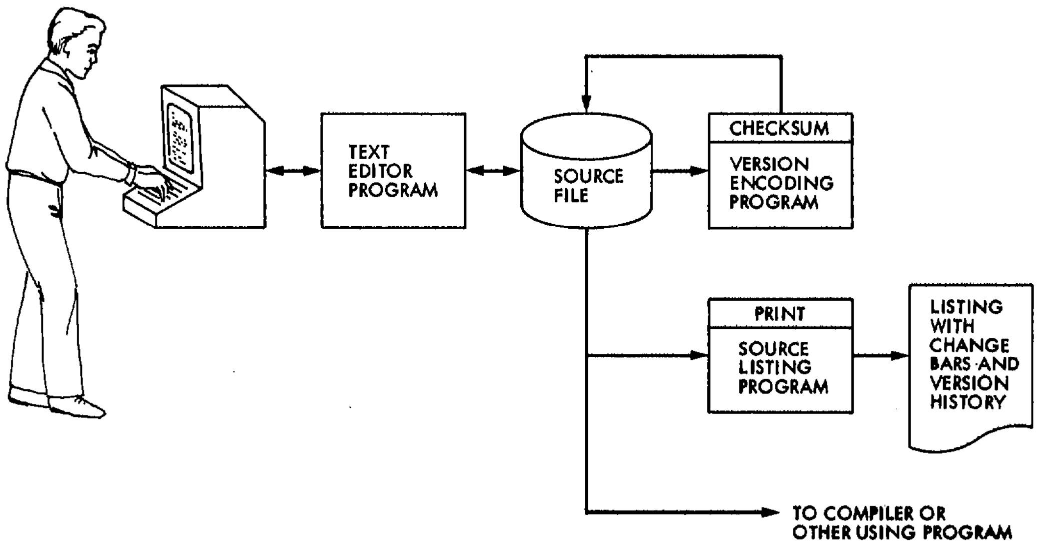

Fig. 1. Software development version control and change identification scenario

```
Program: CHECKSUM
/*************************************************************/
/*                                                           */
/* This program inserts a version code on each physical line */
/* of a source file for use in applications where version    */
/* statistics of such files are to be extracted and analyzed. */
/*                                                           */
/*************************************************************/

.1     define global_constants
.2     declare global_variables
.3     initialize the program
.4     announce the program title and version
.5     get_todays_date /* encode it into date_code */
.6     loop /* for at least one file, and maybe more */
.7       !  get_file_names(IN_FILE, OUT_FILE)
         !    /* OUT_FILE is a dummy file name only */
.8       !  open both files
.9       !  set_up_version_delimiters /* using extension from */
         !    /*  IN_FILE name to identify the particular     */
         !    /*  delimiters.                                 */
.10      !  reset_parameters /* to initial values, in case    */
         !    /* more than one file is processed.            */
.11      !  loop while not end_of_file on input
.12      !    !  input a line /* check for ASCII characters   */
         !    !    /* only and set WARNING switch if not      */
         !    !    /* already set and file is not ASCII       */
.13      !    !  extract_version_code from input line
.14      !    !  compute_checksum of input line
.15      !    !  reattach_new_version_code
         !    !    /* add tabs to code_column, if indicated
         !    !    /* also, set WARNING if line would be too long
.16      !    !  output the line
         !    !..repeat
.17      !  close files
.18      !  if WARNING not set
.19      !  :    rename the OUT_FILE as IN_FILE
         !  :->(else)
.20      !  :    inform the user that the checksummed file has
         !  :      the dummy file name
         !  :..end if
.21      !..repeat if user desires another run
       end program
```

Fig. 2. The main program design of the CHECKSUM program

```
Data GLOBAL_CONSTANTS

     ...

Let FILE_INFO be a table containing delimiter information vs file type
  Each "record" of the table has the following format:
    .  FILE_EXTENSION, the search key which determines the
           following:
    .  KIND, integer, a composite made up of SourceMode and PDLmode:
        SourceMode = KIND modulo 3
                   = 0 for left delimiter required only in source file
                   = 1 for left and right delimiters both required
                   = 2 for version code located by column only
        PDLmode    = (KIND/3) - 1
                   = -1 when no CRISP extraction is to take place
                   = 0 for left delimiter required only in source file
                   = 1 for left and right delimiters both required
                   = 2 for version code located by column only
    .  EXTRACT_SIGNAL, string prefixing lines that are embedded PDL
    .  CODE_COLUMN, integer, for placement of code on output line, when >0
    .  LEFT_DELIMITER[0]  for source lines that are not PDL
    .  RIGHT_DELIMITER[0]  "    "      "      "   "   "   "
    .  LEFT_DELIMITER[1]  for source lines that are PDL comments
    .  RIGHT_DELIMITER[1]  "    "      "      "   "   "   "
                          /* right delimiter may be null string */

     ...

     end data
```

Fig. 3. Excerpt of data definition showing FILE-INFORMATION table format

```
To EXTRACT_VERSION_CODE

.1      set BEGINNING to the first non-blank character after
          the line number field on the input line (if any)
.2      if (the line begins with the EXTRACT_SIGNAL)
        :    set EXTRACT = 1
        :->(else)
        :    set EXTRACT = 0
        :..end if
.3      increment SEQUENCE_NUMBER[EXTRACT] /* This is the      */
          /* sequence number that should be found in the      */
          /* version code of the incoming line                */
.4      set SEQUENCE_CODE = VERSION_CODE(SEQUENCE_NUMBER[EXTRACT])
          /* this is the sequence number code that will be    */
          /* put on the output line                           */
.5      compute where version code would begin on input line,
          if it were present
.6      if (EXTRACT > 0)
.7      :    set MODE = PDLmode
        :->(else)
.8      :    set MODE = SourceMode
        :..end if
.9      case (MODE)
        :->(0) /* Left delimiter only in version code         */
.10     :    if (LEFT_DELIMITER[EXTRACT] is at proper place
        :    :  on input line) then
.11     :    :    extract 2-character CHECK_CODE and OLD_DATE_CODE
        :    :      from input line
        :    :->(else)
        :    :    set LINE_CODE = NULL
        :    :..end if
        :->(1) /* Both left and right delimiters in version code /*
.12     :    if (LEFT_DELIMITER[EXTRACT] and RIGHT_DELIMITER[EXTRACT]
        :    :  at proper place on input line) then
.13     :    :    extract 2-character LINE_CODE and OLD_DATE_CODE
        :    :      from input line
        :    :->(else)
.14     :    :    set LINE_CODE = NULL
        :    :..end if
        :->(2) /* Version code starts in CODE_COLUMN of input line */
        :    extract 2-character LINE_CODE and OLD_DATE_CODE
        :      from input line
        :..end case
       end to
```

Fig. 4. The procedure for extracting the CHECK-CODE and OLD-DATE-CODE from the input line and computing the proper sequence number for the line

```
To COMPUTE_CHECKSUM

.1      if (EXTRACT > 0) /* i. e., if there is an              */
        :  /* EXTRACT_SIGNAL on the line                         */
.2      :    set BEGINNING = BEGINNING + length(EXTRACT_SIGNAL)
        :      /* so as to jump over the extraction signal in   */
        :      /* the checksum count.                           */
        :..end
.4      set EXPANSION COUNT = 0
        LINE_SUM = 0
.5      loop for i = 1 to length of input line
.6        !  set CH to the i-th character on the input line
.7        !  if (i >= BEGINNING)
.8        !  :    set LINE_SUM = LINE_SUM XOR 2*(LINE_SUM AND 16383);
          !  :        LINE_SUM = LINE_SUM XOR (LINE_SUM/16384) XOR CH
          !  :      /* XOR and AND operate on the binary         */
          !  :      /* equivalents of each operand.              */
          !  :..end if
.9        !  if CH is a tab (control-I)
.10       !  :    increment EXPANSION_COUNT by
          !  :      TAB_WIDTH-[(EXPANSION_COUNT-1) modulo TAB_WIDTH]
          !  :      /* to count the equivalent column that the */
          !  :      /* output line will be in at this point    */
          !  :->(else)
.11       !  :    increment EXPANSION_COUNT only by 1
          !  :..end if
          !..repeat
.12      set LINE_SUM = (LINE_SUM MOD 8281);
             CHECK_CODE = VERSION_CODE(LINE_SUM)
         end to
```

Fig. 5. The procedure for computing the checksum code and column at end of input line

```
To REATTACH_NEW_VERSION_CODE

.1    print "." on the terminal to indicate a line has been processed
.2    if (CODE_COLUMN >0) /* i. e., the code is supposed to be */
      :  /* put in a certain column                             */
.3    :    NumberTabsNeeded = max[(CODE_COLUMN + TAB_WIDTH -2
      :                       - EXPANSION_COUNT)/TAB_WIDTH, 0]
      :->(else)
.4    :    NumberTabsNeeded =0
      :..end if
.5    lop the old version code and any trailing spaces and tab
        characters off of the input line
.6    if (the line would be too long if a version code were appended)
.7    :    print a warning message
.8    :    set WARNING = TRUE
      :->(else)
.9    :    make output line by concatenating input line with
      :      NumberTabsNeeded tab characters, LEFT_DELIMITER[EXTRACT],
      :      CHECK_CODE, and SEQUENCE_CODE
.10   :    if (LINE_CODE = CHECK_CODE)
.11   :    :    append OLD_DATE_CODE to output line
      :    :->(else)
.12   :    :    append TODAY_CODE to output line
      :    :..end if
.13   :    finally, append RIGHT_DELIMITER[EXTRACT] to output line
      :      /* It may be null */
.14   :    if (LINE_CODE <> CHECK_CODE)
.15   :    :    print the output line on the terminal
      :    :..end if
      :..end if
      end to
```

Fig. 6. Procedures for reattaching the version code to the output line

```
Program: CHECKSUM                                                  {hT  (r}
/***************************************************************/ {RI !(r}
/*                                                             */ {u# "(r}
/* This program puts a version code on each physical line      */ {f% #(y}
/* of a source file for use in applications where version      */ {7e $(r}
/* statistics of such files are to be extracted and analyzed. */ {VQ %(r}
/*                                                             */ {u# &(r}
/***************************************************************/ {RI '(r}
                                                                   {   ((r}
.1     define program_constants                                    {CF ))3}
.2     declare program_variables                                   {JU *)%}
.3     set_initial_values                                          {N/ +(y}
.4     announce the program title and version identifier          {"V ,(y}
.5     get_todays_date /* encode it into date_code */             {$n -(r}
.6     loop /* for at least one file, and maybe more, as will */  {&` .)3}
         ! /* be determined later by the user                 */  {=Q /)3}
.7       ! get_file_names(IN_FILE, OUT_FILE, KIND)                 {p: 0(r}
         !   /* KIND may be input or determined from IN_FILE */
         !   /* OUT_FILE is a dummy file name only            */  {-u 1(y}
.8       ! open both files                                         {SI 3(r}
.9       ! set_up_version_delimiters /* using search key from*/   {kb 4(y}
         !   /*  IN_FILE name to identify the particular     */   {v? 5(y}
         !   /*  delimiters to be used by the program.       */   {U3 6)%}
.10      ! reset_parameters /* to initial values, so that    */   {]P 7)%}
         !   /* more than 1 file can be processed, as will   */   {Y? 8)3}
         !   /* be determined later by the user             ,*/   {=Q /)3}
.11      ! loop while (not end_of_file on input)                   {8W :)3}
.12      !   ! input a line /* check for ASCII characters    */   {b( ;(r}
         !   !   /* only and set WARNING switch TRUE if      */   {K4 <(y}
         !   !   /* file contains non-ASCII characters       */   {"e =)%}
.13      !   ! extract_version_code from input line                {oB >(r}
.14      !   ! compute_checksum of input line                      {c$ ?(r}
.15      !   ! reattach_new_version_code                           {,L @(r}
         !   !   /* add tabs to code_column, if indicated.   */   {"\ A(y}
         !   !   /* also, set WARNING to true if line would */   {Ou B(y}
         !   !   /* be too long if the code were appended.  */   {N^ C(y}
.16      !   ! output the line                                     {'B D(r}
         !   !..repeat                                             {X) E(r}
.17      ! close both files                                        {L\ F(y}
.18      ! if WARNING not set to TRUE                              {Jj G(y}
.19      ! :    rename the OUT_FILE as IN_FILE                     {3d H(r}
         ! :->(else)                                               {eJ I(r}
.20      ! :    inform the user that there has been a problem,    {v5 J(y}
         ! :      and that the dummy is the checksummed file      {\[ K(y}
         ! :..end if                                               {%H M(r}
.21      ! prompt user for "what next", and accept answer         {UE N(y}
.22      !....repeat if user desires another run                   {>a O(y}
       end program                                                 {rJ P(r}
```

Fig. 7. The CHECKSUM program design language file after execution of the CHECKSUM program on the file

```
                    FILE:  CHECKSUM.PDL,  STATUS:  26MAR82
Program: CHECKSUM
/**************************************************************/
/*                                                            */
/* This program puts a version code on each physical line     */
/* of a source file for use in applications where version     */
/* statistics of such files are to be extracted and analyzed. */
/*                                                            */
/**************************************************************/

.1     define program_constants                                      []
.2     declare program_variables
.3     set_initial_values
.4     announce the program title and version identifier
.5     get_todays_date /* encode it into date_code */
.6     loop /* for at least one file, and maybe more, as will */     []
         !  /* be determined later by the user                */     []
.7       !  get_file_names(IN_FILE, OUT_FILE, KIND)                  [M]
         !    /* KIND may be input or determined from IN_FILE */     [A]
         !    /* OUT_FILE is a dummy file name only           */
                                                                     [D]
.8       !  open both files
.9       !  set_up_version_delimiters /* using search key from*/
         !    /*  IN_FILE name to identify the particular     */
         !    /*  delimiters to be used by the program.       */
.10      !  reset_parameters /* to initial values, so that    */
         !    /* more than 1 file can be processed, as will   */    [M]
         !    /* be determined later by the user              */   [RM]
.11      !  loop while (not end_of_file on input)                    []
.12      !    !  input a line /* check for ASCII characters   */
         !    !    /* only and set WARNING switch TRUE if    */
         !    !    /* file contains non-ASCII characters     */
.13      !         !  extract_version_code from input line
.14      !    !       compute_checksum of input line
.15      !    !  reattach_new_version_code
         !    !    /* add tabs to code_column, if indicated.  */
         !    !    /* also, set WARNING to true if line would */
         !    !    /* be too long if the code were appended.  */
.16      !    !  output the line
         !    !..repeat
.17      !  close both files
.18      !  if WARNING not set to TRUE
.19      !  :    rename the OUT_FILE as IN_FILE
         !  :->(else)
.20      !  :    inform the user that there has been a problem,
         !  :      and that the dummy is the checksummed file           [M]
                                                                     [D]
         !  :..end if
.21      !  prompt user for "what next", and accept answer
.22      !....repeat if user desires another run
       end program
```

Fig. 8. The output of the PRINT program, showing changes made in the CHECKSUM program design since March 26, 1982

```
V E R S I O N   S T A T I S T I C S

 Number of source statements =  48

   Since last version update:

 new (no vers id) statements =   1

 apparent deleted statements =   3

         modified statements =   3

       duplicated statements =   1

 Other mod since status date =   4

     C H A N G E   H I S T O R Y


              Date No.
             05MAR82  20
             12MAR82  15
             19MAR82   4
             02APR82   4
             16APR82   5
```

Fig. 9. The Version Statistics and Change History printout for the CHECKSUM program design